Enhancing DEVS Simulation through Template Metaprogramming

Luc Touraille, Mamadou Kaba Traoré, David R.C. Hill

Enhancing DEVS Simulation through Template Metaprogramming

DEVS-MetaSimulator

Luc Touraille[1], Mamadou K. Traoré[2], David R.C. Hill[3]

Research Report LIMOS/RR-10-12

19 mai 2010

[1] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, touraille@isima.fr

[2] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, traore@isima.fr

[3] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, drch@isima.fr

Abstract

For several years, the DEVS community has been developing many tools for simulating DEVS models, ranging from local sequential to massively distributed and parallel simulation. In this paper, we present an innovative approach to local DEVS simulation. By using template metaprogramming, we developed the DEVS-MetaSimulator (DEVS-MS); instead of proposing one simulator meant to be used with every DEVS models, our library provides several metaclasses defining families of simulators. This way, each simulator instantiation is really specialized for a particular model. Doing so, we increase the detection of errors at compile-time, and we greatly reduce the execution time by removing several runtime computations that are instead performed by the compiler.

Keywords: DEVS, template metaprogramming

Résumé

Depuis plusieurs années, la communauté DEVS a développé de nombreux outils pour la simulation de modèles DEVS, allant de la simulation séquentielle et locale à la simulation parallèle et massivement distribuée. Dans cet article, nous présentons une approche innovante pour la simulation DEVS locale. En utilisant la métaprogrammation template, nous avons développé le DEVS-MetaSimulator (DEVS-MS) ; au lieu de proposer un simulateur unique pour l'ensemble des modèles DEVS, notre librairie fournit plusieurs métaclasses qui définissent des familles de simulateurs. Chaque instance de simulateur est ainsi entièrement spécialisée pour un modèle donné. De cette façon, nous augmentons la détection d'erreurs à la compilation et réduisons le temps d'exécution en effectuant plusieurs opérations de simulation à la compilation plutôt qu'à l'exécution.

Mots-clés : DEVS, métaprogrammation template

1. INTRODUCTION

Since its definition, the Discrete EVent System Specification (DEVS) [14] [15] formalism has been studied by an important community of scientists. Many tools have been developed to create and simulate DEVS models [4] [5] [6] [7] [8] [13].

However, due to the nature of the formalism, these applications often lack certain security guarantees, notably type safety. Indeed, messages exchanged between DEVS models can be of potentially any type. Therefore, the simulator must be able to deal with every possible type, most of the time resulting in breaches in the type system due to hazardous type casting, possibly generating hard to detect runtime issues. Other subtle errors can be introduced due to the extensive use of strings to represent the different model entities.

In this paper, we present a DEVS MetaSimulator (DEVS-MS) that resolves these issues by using a programming technique called template metaprogramming.

Template metaprogramming not only provides complete type safety and name checking, it can also be leveraged to carry out several computations at compile-time [1] [10] [11]. In this paper, we show that many simulation operations, usually made during execution, can be performed beforehand, during compilation, resulting in increased performance and robustness of the final simulation program at runtime.

2. DEVS FORMALISM

The Classic DEVS formalism, proposed by Zeigler in the mid-seventies [14], defines two kinds of models for representing a system. An *atomic model* is an entity holding a state and evolving via internal state changes and responses to external stimuli. After undergoing an internal event, the model generates an output event that is propagated to the outside. Formally, a DEVS atomic model is defined by a 7-uplet:

$$M = \langle X, Y, S, ta, \delta_{int}, \delta_{ext}, \lambda \rangle$$

where:

- X is the set of inputs. Usually, X is decomposed into several sets representing model *ports*. For example, to model a lake that can receive a certain amount of pollutant and be repopulated with newly-hatched fishes (alevins), we could define the input set by
  $X = \{ (pollutant, alevins) \mid pollutant \in \mathbb{R}, alevins \in \mathbb{N} \}$
- Y is the set of outputs. It is defined similarly to X, using ports.
- S is the state set, containing all the possible characterizations of the system.
- ta is the time advance function. It determines how long the system stays in a given state before undergoing an internal transition and moving to another state.
- δ_{int} is the internal state transition function. It defines how the system evolves in an autonomous way.
- δ_{ext} is the external state transition function. It defines how the system reacts when stimulated by an external event.
- λ is the output function. Invoked after each internal transition, it determines the events generated by the system, functions of the state it was in.

Atomic models can be combined to form *coupled models*, which can be in their turn put together, leading to a hierarchic model. A coupled model aggregates components, either atomic or coupled models, linked by their input and outputs ports; thus, the events generated by a model become stimuli for others. Formally, a DEVS coupled model is characterized by a structure:

$$N = \langle X, Y, D, \{M_d \mid d \in D\}, \{I_d\}, \{Z_{i,d}\}, Select \rangle$$

where:

- X is the set of input ports and values, in the manner of atomic models.
- Y is the set of output ports and values.
- D is the set of component names.
- For each d in D, M_d is a DEVS model (coupled or atomic).
- I_d is the influencer set of d, i.e. the names of the components that impact d, through their output ports.
- $Z_{i,d}$ is the coupling function. It maps N input ports to component input ports, component output ports to component input ports, and component output ports to N output ports.
- *Select* is the tie-breaking function. Classic DEVS is fundamentally sequential. Therefore, when two models are supposed to undergo an internal transition – and consequently generate an event –, only one of them must be activated. This arbitrage is performed via the *Select* function.

The operational semantic of these models is defined by Zeigler [15] through abstract simulators, meaning algorithms that correctly simulate DEVS models. The simulation software presented in this paper, DEVS-MS, implements these algorithms in an innovative way, using several metaprogramming techniques.

3. RATIONALE

The main idea behind DEVS-MS is to provide a framework where each coordinator and simulator[4] would be specialized for the model it handles. To illustrate this, we will consider the classic Switch Network coupled model, proposed by Zeigler in [15] and described in **Erreur ! Source du renvoi introuvable.**.

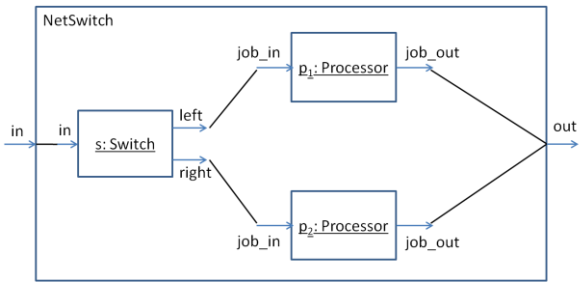

Figure 1. Switch network coupled model.

This model is composed of three components: a switch *s* that forwards its input alternatively to its "left" port and "right" port, and two processors p_1 and p_2, which accept an input job, await some time and output the job. In our example, we will assume that jobs are represented by strings.

Now imagine that we could have one specialized simulator for each component: as presented in Code 1, we could end up with a much more secure and efficient simulator. In this code, which presents potential code snippets, we call `sim_s`, `sim_p1` and `sim_p2` the simulators for `s`, `p1` and `p2` respectively.

The simulator of *s* has two methods, one for receiving an event on the left port, and the other one for the right port. These member functions accept a string, based on the port types of Switch. Since the simulator is specialized for *s*, it can directly forward the events to the correct simulators, i.e. `sim_p1` and `sim_p2`, relieving the coordinator from the burden of computing the couplings at runtime.

```
class Switch
{
  ...
  // In this case, sim would refer to sim_s
  void lambda()
  {
    if (...)
      sim.putOnLeft(output);
    else
      sim.putOnRight(output);
  }

  string output;
};

class SimulatorSwitch_s
{
  ...
  void putOnLeft(string output)
  {
    sim_p1.putOnJob_in(output);
  }

  void putOnRight(string output)
  {
    sim_p2.putOnJob_in(output);
  }
};
```

Code 1: General switch model and specialized switch simulator.

[4] In the rest of the document, we will refer to simulators and coordinators as *model handlers*, or simply *handlers*.

Similar things could be done in the coordinator of the network switch: for example, the Code 2 shows how we can process the model internal transitions when we have an a priori knowledge of the components and their priorities (i.e. the Select function). Here, p_1 always take priority over p_2, which always take priority over s.

```
class CoordinatorNetSwitch
{
    ...
    void processInternalTransitions(time t)
    {
        if      (sim_p1.tn == tn)  sim_p1.processInternalTransition(t);
        else if (sim_p2.tn == tn)  sim_p2.processInternalTransition(t);
        else                       sim_s.processInternalTransition(t);

        tl = t;
        tn = min(sim_p1.tn, sim_p2.tn, sim_s.tn);
    }
}
```

Code 2: Specialized switch network coordinator.

Existing DEVS simulators, whether they are written in C++ [5] [7] [13], in Java [4] [8] or in Python [3], propose "one-fits-all" coordinator and simulator. The only exception is the ADEVS simulator [6], where models are parameterized by their input and output types; thus, simulators and coordinators know the type of message they should receive.

In DEVS-MS, we provide an endless number of specialized model handlers. Indeed, we developed several meta-classes that define entire families of simulators and coordinators. When such a meta-class is instantiated with a particular model, the compiler generates a specialized version of the corresponding handler, along the lines of the code presented above. To achieve this, we use some metaprogramming techniques that are described in the next section.

4. METAPROGRAMMING

4.1 Presentation

The simplest definition of a metaprogram is "a program that creates or manipulates one or more other programs". Very broad, this definition embraces many different things, such as reflection, runtime expression evaluation or code generation. In our work, we used a particular technique called Template MetaProgramming (TMP hereafter) [1] [2] [10] [11], and more particularly C++ TMP[5].

The main idea behind TMP is to make the compiler execute certain operations during compilation through template instantiation. The most common use is code generation, notably in generic programming, where data structures and algorithms are written once and for all but can work with any type meeting a small set of requirements (e.g. supporting the dereferencing operator '*'). However, the template mechanism can be further leveraged to make the compiler perform several computations at compile-time instead of runtime. The canonical example is the factorial: as described in Code 3, we can use recursion and template specialization to compute the value of a factorial during compilation, reducing the execution time (but of course increasing the compilation time).

```
template <unsigned char N>
struct Factorial
{
  static const unsigned long result =
    N * Factorial<N-1>::result; // recursive call
};

template <>
struct Factorial<0> // base case
{
  static const unsigned long result = 1;
};

int i = Factorial<5>::result; // int i = 120;
```

Code 3. Factorial computation using TMP

[5] Some other languages support or are extended to support TMP; however, C++ remains in this domain the most developed, documented and used one.

As a matter of fact, templates have been proven to be Turing complete [10]; however, the range of computations that can be practically written is narrower. Most of them deal with type manipulation, as we'll see in the next paragraph.

4.2 Boost libraries

Boost is a set of free peer-reviewed C++ libraries, most of them being highly regarded by the community. In order to facilitate metaprogramming, we extensively used four of them in our work: TypeTraits, MetaProgramming Library (MPL), Fusion and Preprocessor.

4.2.1 Type-traits

The type-traits library contains a set of convenience classes for manipulating types. Each of them encapsulates a specific trait from the C++ system; for example, there are classes for determining whether a type is an integral type, whether it is const, whether it has a trivial constructor, etc. The library also contains some classes for performing very simple type transformations, e.g. adding a reference or removing a pointer to a type.

The classes of that kind represent functions invocable at compile-time, and hence are called *metafunctions*. Calling a metafunction boils down to instantiating a class template and if need be getting the result by accessing a typedef type in it. The metafunctions proposed by the type-traits library are rather simple, but they are essential primitives[6] to create more complex frameworks, such as the MPL or the Fusion library.

4.2.2 MetaProgramming Library (MPL)

The MPL is a metaprogramming framework; in the manner of the Standard Template Library (STL), it provides the user with a set of sequences, algorithms and metafunctions for doing compile-time computation.

MPL sequences supply an easy way to create, transform and query compile-time sequences of types. For example, the following code defines a sequence containing the types `int`, `float` and `char`:

```
typedef boost::mpl::vector<int, float, char> seq;
```
[7]

Since `vector` is a model of the Random Access Sequence concept, we can use the `at_c` metafunction to retrieve the nth element in the sequence, in amortized constant compilation time:

```
mpl::at_c<seq, 1>::type f1 = 0.0f; // <=>
float                   f2 = 0.0f;
```

Now, assume that we need to transform this sequence so that it contains only integral types. To do so, we can use the `remove_if` algorithm, a higher-order metafunction that removes all the elements in a sequence that satisfy a given predicate:

```
typedef mpl::remove_if<
    seq,                            // sequence
    mpl::not_<
      boost::is_integral<mpl::_>
    >                               // predicate
>::type int_seq;                    // result

//int_seq is equivalent to mpl::vector<int, char>
```

MPL proposes many sequences (`list`, `map`, `set`…), metafunctions (`at`, `begin`, `end`…) and algorithms (`find`, `copy_if`, `transform`…). Most of them are very similar to their STL counterpart, from the way they are used to their complexity guarantee, except that instead of dynamically dealing with sequences of values at runtime, they statically deal with sequences of types at compile-time.

4.2.3 Fusion

As we explain above, MPL provides a set of powerful tools for performing operations on sequences of types, during compilation. However, at some point in time, we will need to map these types to values to make them

---

[6] Actually, the C++ committee found them useful enough to integrate them into the next ISO C++ standard (C++0x).

[7] In the rest of the document, we'll assume the namespace `boost::mpl` has been aliased to `mpl`., and `boost::fusion` to `fusion`.

useful in the runtime world. That's where Fusion comes into play: this library aims at uniting compile-time metaprogramming with runtime programming.

Concretely, the library provides a set of heterogeneous containers (tuples) that can hold elements with arbitrary different types, along with several functions and algorithms to operate on them. Each operation has its "meta" counterpart, which performs the same kind of computation at compile-time. For example, the `begin` function will return the first element of a sequence, while the metafunction `result_of::begin` will return *the type* of the first element in a sequence. However, the metaprogramming aspect is a bit more tedious in Fusion than in MPL; a common practice is to do all the pure type calculations using MPL, then instantiate Fusion sequences with the results of these calculations.

4.2.4 Preprocessor

Unlike the three libraries presented previously, the Preprocessor library does not provides tools for manipulating types, but a set of macros for making the C++ preprocessor generates repetitive code that would be hard and painful to come up with if it was to be written by hand.

A classic (and – for once – really useful) example is overcoming the lack of variadic templates in C++. Especially when doing metaprogramming, we are often confronted with the need to write a class or function template that can accept an arbitrary number of parameters. Take the `mpl::vector` class evoked before: the number of template parameters it should be able to deal with ranges from zero to theoretically infinity. One solution would be to define the class template with a great number of defaulted template parameters. However, this is not always practical or even possible. Consequently, we end up having to define multiple classes or functions that are almost identical, which is a tedious and error-prone task.

The Preprocessor library tackles this issue by proposing a set of macros for performing repetitions (repeatedly invoking a given macro with an incrementing argument – think for loop), arithmetic, logical and comparison operations and conditional macro invocation. It even defines several data types for storing sequences of macro arguments.

5. IMPROVING MODEL DEVELOPMENT AND SECURITY CHECKS

The first improvements that can be achieved using TMP concerns model implementation. The aim is to prevent the modeler from making certain accidental mistakes by detecting them as soon as possible in the development cycle, i.e. during compilation.

5.1 Ports compatibility (type safety)

The DEVS formalism is very generic, and does not define any constraints on the type of model inputs and outputs. Consequently, the value associated with some event can be of potentially any type. The simulator must be able to deal with this heterogeneity and provide a generic way of handling messages. To achieve this, the classic polymorphic approach is to require a common base class for all the types used in messages. However, this solution has many drawbacks.

First of all, this constraint prevents the modeler from using certain types, like primitive types that are not classes (e.g. `int`, `char`... in Java and C++) or existing classes that cannot be modified to integrate the hierarchy (e.g. C++ `string`). Often, wrappers are proposed to encapsulate this kind of value into objects: in Java, they are supported directly by the language, in other languages they are written by the library authors. However, using these wrappers induces a small runtime overhead due to the conversions, but more importantly they do not solve the true issue behind the polymorphic solution: type-safety.

Indeed, the use of a "universal" base class completely defeats the purpose of the language type system. After being upcasted, the values become of little use; they need to be downcasted back to their original type before being really exploited. If the cast is not correct, a runtime error will be produced. This can happen when coupled ports are not compatible, for example coupling a model outputting a string with a model accepting an integer. These errors can be due to a mistake from the modeler, but can also happen when using imported models that can undergo future changes, possibly breaking their interface.

The solution we developed uses TMP to achieve static typing of messages. To do so, we created a class template Message, parameterized by an MPL sequence of types[8]. When constructed, objects of this class instantiate a

[8] In Classic DEVS, a message represents only one event on one port. However, we wanted to be able to reuse the Message class for Parallel DEVS; this is why it accepts a sequence of types instead of a single one.

Fusion sequence that will hold the values carried out by the message. This way, type information is never lost and message exchanges are guaranteed to be type-safe. The compiler finds the inconsistencies and produces errors messages underlining the incompatibility between the incriminated ports. In case of problems, the modeler is warned much sooner in the development process, without having to run test suites to explore every possible case (or worse, perform no tests and stumble upon strange errors at a later point in time).

5.2 Names checking

Another improvement of our DEVS simulator concerns names. In Classic and Parallel DEVS, names used throughout a model are not meant to be modified during simulation. Once a port or a component has been identified, its designation will not change. Yet, names are usually represented by strings, which are runtime values. Once again, this leads to additional computations, notably string comparisons; but more important, the compiler has no way to check whether the names are consistent in the entire model. If the modeler makes a mistake when reusing a name, for example when defining the couplings or when writing lambda functions, he will not know until he executes the simulation. If he is lucky, the error will show up pretty soon, but if the mistake was made in a part of the model which is scarcely used, it can keep buried a long time before being revealed and corrected.

Once again, this can be solved using TMP. The main idea is to use types to denote ports and components instead of using strings. For example, if a model contains a port named "in", we will create a type `in`:

```
struct in;
```

From now on, each time we will need to refer to this port, we will use this type. This leads to some changes in the Message class presented above: instead of being parameterized by a sequence of types, it will accept a sequence of pairs. Each pair contains the name of the port (as a type), and the type of value it accepts/generates. To facilitate the use of this information, we store the ports in an MPL map whose key is the port name and value the type of the port. Runtime values are held in a Fusion map which statically associates port names with their corresponding value.

The C++ Code 4 presents a sample member function of the Message class. We followed the example of the Fusion library by proposing each functions in two flavors: compile-time and runtime. Here, the compile-time version of GetValueOnPort is a metafunction returning the type held on a given port, whereas the runtime version returns the actual value.

An important fact to stress is that accessing values this way is equivalent to accessing data members of the class, all the map operations are performed at compile-time so no runtime overhead occurs.

As we will see in section 6, using types for representing names not only provides a better model verification, it also opens the doors to many important optimizations. Another aside benefit concerns the ease of development: instead of having to remember every single name used in the model, the user can rely on its IDE code completion tool.

```
template <typename Ports>
class Message {

  // We create two maps for ports, one static and
  // one dynamic:
  // - an MPL map holding (name, type) pairs
  // - a Fusion map holding (name, value) pairs

  ...
  struct result_of {
    template <typename PortName>
    struct GetValueOnPort
    {
      // Returns the type held by PortName
      typedef
        mpl::at<mplPortsMap, PortName>::type type;
    };
  };

  template <typename PortName>
  result_of::GetValueOnPort<PortName>::type
  getValueOnPort() const
  {
    // Returns the value on PortName
    return fusion::at_key<PortName>(ports);
  }
};

Message<mpl::vector<mpl::pair<in, float> > > m;
float f = m.getValueOnPort<in>();
```

Code 4: Small part of the Message class template.

5.3 Additional checks

Still in the perspective of preventing accidental mistakes, we implemented several other consistency checks. For example, when a simulator receives a message from its atomic model, it checks that this message is consistent with the output ports defined for the model. We achieve this by requiring that ports (i.e. pairs of name and type) be explicitly specified in the model through a typedef, along other information such as the type of the model, using a tag (atomic or coupled). To release the modeler from the burden of writing himself all these typedefs, we created a class template for coupled and atomic model. Therefore, the user only has to derive his model from one of these templates with the proper parameters, meaning the sequence of input and output ports, to automatically inherit the required typedefs. With this information, we can statically (at compile-time) assert that messages outputted are consistent with the definition of the model ports.

We also ensure that simulators and coordinators are properly used by "specializing" them according to the inputs and outputs of models. Concretely, we remove functions that shouldn't be called in a correct simulation. Thus, the handler of a model with no input ports will not have any function for handling input messages; similarly, if an atomic model has no outputs ports, its simulator will not expose a function for receiving internal events, nor call the model lambda function.

6. IMPROVING SIMULATION PERFORMANCES

We saw in the previous section that using TMP, we increased the robustness of the simulation by making the compiler performs several consistency checks. In the same time, we also introduced optimizations, for example removing the need for all the type casting. We will see now that we can make the most of the metaprogramming approach to further increase simulation performances, notably by computing several simulation operations at compile-time.

6.1 Static polymorphism

The first impact on performances is a direct consequence of the developments presented above. As we said, the only requirements on models are providing the needed typedefs, and of course the mandatory members functions required by the formalism. Therefore, models can be completely independent classes, and do not necessarily share a common base class. In fact, even if the modeler decides to inherit from the coupled or atomic model classes proposed in the library, they will still be unrelated because the instantiated classes will be different. As a consequence, simulators and coordinators need to be parameterized by the type of the model they handle. This

method is called static polymorphism, in opposition with the more classic dynamic polymorphism built-in all object-oriented languages.

Using static polymorphism in our simulation framework induces several advantages: first, the handlers obtain useful compile-time information that will be later exploited; second, no more virtual function calls are needed. Doing so, we removed a layer of indirection for calling functions and allowed the compiler to apply inline expansion[9] where it sees fit, both consequences leading to a decrease of runtime overhead.

6.2 Computing the select function

In Classic DEVS, the Select function of coupled models is defined once and for all and thus can be encoded in a compile-time constant. We represent the select function by an ordered set containing ordered sets of component names. When several components should undergo their internal transition at the same time, we look up in the Select set the first list including all the candidate names. The first element of this list has the highest priority and will be activated.

Obviously, the set of imminent components need to be computed at runtime since it depends on their time of next event. On the other hand, the choice of the component to activate can be made by the compiler. We achieve this by intertwining compile-time and runtime parameters. This is exposed in the pseudo-code Code 5.

```
function internal_transition
  < >
  ( )
{
  call recursive_filter
         < [], ChildrenMap.Begin >
         ( children.begin );
}

function recursive_filter
  < [ci, ..., cj], It == ChildrenMap.End >
  ( It end )
{
  call star_message_continuation
         < [ci, ..., cj] >
         ( );
}

function recursive_filter
  < [ci, ..., cj], It != ChildrenMap.End >
  ( It it )
{
  if (it.tn == tn)
    call recursive_filter
           < [ci, ..., cj, It.name], It++ >
           ( it++ );
  else
    call recursive_filter
           < [ci, ..., cj], It++ >
           ( it++ );
}

function internal_transition_continuation
  < [ci, ..., cj] >
  ( )
{
  Look up [ci, ..., cj] in Select
  SelectedName  := the first name in this set
  SelectedChild := the handler of ActivableName

  call selectedChild.internal_transition<>();
}
```

Code 5: Component activation: runtime filtering and compile-time selection

[9] *Inline expansion*, or *inlining*, is a compiler optimization where a function call is replaced by the actual body of the function, avoiding several stack operations and making possible further optimizations.

Each function has two kinds of parameters: compile-time parameters, which are enclosed in brackets, and runtime parameters, between parentheses. To filter the imminent components, we call the `recursive_filter` function with an empty sequence, which will contains the names of the candidate components, and an iterator pointing to the beginning of the components map (components are stored in a Fusion map that associates their name with the actual instance). This function is specialized to handle the end of recursion: if we reach the end of the components list, then we "statically return" the list of candidate components. Otherwise, we test whether the current component is ready to perform its next transition. In this case, we recursively call `recursive_filter` with the input sequence of candidates, completed with the component name, and the incremented iterator. If not, we don't modify the sequence of imminent components and only increment the iterator. The `star_message_continuation` function "statically receive" the list of candidates and select the correct component, which will be activated at runtime.

6.3 Forwarding messages

The select function is not the only constant piece of information we can exploit. Indeed, except in Dynamic Structure DEVS, the couplings between components remains the same during the entire simulation. Consequently, we can check couplings and forward messages to the influenced components by means of the compiler.

In DEVS-MS, couplings are defined for a coupled model via an MPL sequence. Each coupling in this list is a pair containing two pairs: the source component name and port, and the destination component name and port. We statically transform this sequence into an MPL map, in order to facilitate the look-up when forwarding messages.

When receiving an event, the coordinator can deduce from its type the set of influenced components, based on the ports list of the message. It can also carry out the port mapping between output and input ports, always at compile-time. Therefore, the only operations that remain to be done dynamically are sending the messages to the corresponding simulators. Assuming some aggressive inlining from the compiler, we can end up with simulators – or even components – exchanging messages directly, without going up the coordinators hierarchy, as described in section 3.

6.4 Filtering input events

As we saw previously, messages encapsulate information about the ports they concern. In the case of an input event, the message contains the name of the input port he is sent to. We can make use of this knowledge to perform static filtering of messages in the destination model. Because they are instantiated classes, messages heading to different ports are of different types. Thanks to this property, the modeler can overload the model external transition function to segregate the different cases:

```
void delta_ext(const Message<Job>    & x) {...}
void delta_ext(const Message<On_off> & x) {...}
```

The choice of the correct function will be made through overload resolution by the compiler.

In fact, compile-time filtering will happen even if the modeler decides to write a unique external transition function with a nested if:

```
template <typename Msg>
void delta_ext(const Msg & x)
{
  if ( x.hasValueOnPort<Job>() ) ...
  else if ( x.hasValueOnPort<On_off>() ) ...
}
```

Here, current compilers are smart enough to detect that only one branch will always be true, and simply remove the test and the other branches.

7. Conclusion and future works

We proposed an innovative approach to DEVS simulation. By using a technique called Template MetaProgramming, we developed a simulator that makes extensive use of the compiler for two tasks: removing many potential bugs as early as possible in the model development cycle, and increasing simulation performances by transferring certain computations from runtime to compile-time.

To avoid accidental mistakes, we made the compiler performs several consistency checks over the model, most notably verifying the type compatibility of coupled ports, and ensuring the consistency of ports and components

names. These kinds of errors are now caught by the compiler instead of producing bugs, potentially hard to find and correct.

Regarding performances, we took advantage of the constant nature of certain model parts to carry out some simulation operations at compile-time. These include arbitrating the occurrence of simultaneous events through the select function, checking couplings and forwarding messages to influenced components, and filtering input messages by port name.

Our future works will consist primarily in measuring the performance gain on a set of significant examples to compare DEVS-MS with other existing simulators. Then, we may develop versions supporting other formalisms, notably Parallel DEVS. Further developments could also be done to optimize DEVS-MS with regard to compilation time and executable size.

8. REFERENCES

[1] Abrahams, D. and Gurtovoy, A., *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[2] Alexandrescu, A., *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001

[3] Bolduc, J.-S. and Vangheluwe H., *The modelling and simulation package PythonDEVS for classical hierarchical DEVS*. MSDL technical report MSDL-TR-2001-01, McGill University, June 2001.

[4] Filippi, J-B., Bernardi, F. and Delhom, M., *The JDEVS environmental modeling and simulation environment*. Proceedings of the IEMSS'02 Conference on Integrated Assessment and Decision Support. Lugano, Switzerland. 2002.

[5] Kim, T. G., *DEVSim++: C++ based Simulation with Hierarchical Modular DEVS Models*. User's Manual CORE Lab, EE Dept, KAIST, Taejon, Korea. 1994.

[6] Nutaro, J., ADEVS website. Available via http://www.ornl.gov/~1qn/adevs/. Last accessed on September, 2009.

[7] Quesnel, G., Duboz, R. and Ramat E., *The Virtual Laboratory Environment - An Operational Framework for Multi-Modelling, simulation and analysis of complex dynamical systems*. Simulation Modelling Practice and Theory, vol. 17, April 2009, pp. 641-643

[8] Sarjoughian, H. S. and Zeigler, B. P., *DEVSJAVA: Basis for a DEVS-based collaborative M&S environment*. Proceedings of the SCS International Conference on Web-Based Modeling and Simulation, vol. 5, pp. 29--36. San Diego, USA. 1998.

[9] Veldhuizen, T. L. *C++ Templates are Turing Complete*. Technical report, Indiana University, 2003.

[10] Veldhuizen, T. L., *Expression templates*. C++ Report, Vol.7 No. 5, June 1995.

[11] Veldhuizen, T. L., *Using C++ template metaprograms*. C++ Report, Vol. 7 No. 4, May 1995.

[12] Veldhuizen, T.L. and Kumaraswamy, P., *Linear algebra with C++ Template Metaprograms*. Dr. Dobb's Journal. August 1996.

[13] Wainer, G., *CD++: a toolkit to develop DEVS models*. Software—Practice & Experience, vol. 32 n° 13, pp. 1261-1306, 10 November 2002.

[14] Zeigler, B.P., *Theory of Modelling and Simulation*. New York: John Wiley, 1976.

[15] Zeigler, B.P., Praehofer, H. and Kim, T.G., *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*, 2nd Edition. Academic press, 2000.